AFRL-RI-RS-TR-2018-197

TOOLKIT FOR EVOLVING ECOSYSTEM ENVELOPES (TEEE)

ADVENTIUM ENTERPRISES, LLC

AUGUST 2018

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the 88th ABW, Wright-Patterson AFB Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2018-197 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /
STEVEN L. DRAGER
Work Unit Manager

/ S /
JOSEPH A. CAROLI
Acting Technical Advisor, Computing & Communications Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
AUGUST 2018	FINAL TECHNICAL REPORT	AUG 2016 – FEB 2018

4. TITLE AND SUBTITLE

TOOLKIT FOR EVOLVING ECOSYSTEM ENVELOPES (TEEE)

5a. CONTRACT NUMBER
FA8750-16-C-0273

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
61101E

6. AUTHOR(S)

Todd Carpenter, Perry Alexander, Hayley Borck, John Gohde,
Steve Johnston, Paul Kline, Ed Komp, Valerie McKay,
Hazel Shackleton

5d. PROJECT NUMBER
BRAS

5e. TASK NUMBER
SA

5f. WORK UNIT NUMBER
DV

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Adventium Enterprises, LLC
111 Third Ave S., Suite 100
Minneapolis, MN 55401-2551

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RITA
525 Brooks Road
Rome NY 13441-4505

DARPA
675 North Randolph Street
Arlington, VA 22203-2114

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSOR/MONITOR'S REPORT NUMBER
AFRL-RI-RS-TR-2018-197

12. DISTRIBUTION AVAILABILITY STATEMENT

Approved for Public Release; Distribution Unlimited. PA# 88ABW-2018-4031
Date Cleared: 14 AUG 2018

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The primary goal of this project was to create dynamic profiling synthesis tools, models, and protocols to explore Cyber Physical System performance envelopes, subject to their evolving environments, that will ultimately allow software to adapt as internal and external conditions change. The result enables identification and visualization of functional and resource limitations that impact the ability for the system to meet operating requirements within its changed environment.

15. SUBJECT TERMS

Test Generation, Automated Synthesis, Measurement Synthesis

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			STEVEN L. DRAGER
U	U	U	UU	36	19b. TELEPHONE NUMBER (*Include area code*)

Table of Contents

Table of Figures

Table of Tables

Table of Listings

1 Summary

Cyber Physical Systems (CPSs) are software and hardware systems that interact with the physical environment. While many CPSs have useful lifetimes measured in decades, the environments in which these CPSs operate change over time. To remain functional over extended periods, these CPSs must adapt, or be adapted, to these changes. In particular, the software in long-lifetime CPSs must adapt to unanticipated trends in environmental conditions, aging effects on mechanical systems, component upgrades, and component modifications.

This Final Report documents the results of Adventium's Toolkit for Evolving Ecosystem Envelopes (TEEE) project, part of DARPA's Building Resource Adaptive Software Systems (BRASS) program, conducted from August 2016, to February 2018. This program created dynamic profiling synthesis tools, models, and protocols to explore CPS performance envelopes, subject to their evolving environments, that will help adapt software as internal and external conditions change.

CPSs can interact with the physical environment by sensing external state, transferring kinetic and potential energy, computing solutions to affect desired outcomes, and driving electrical, optical, and mechanical actuators to achieve those outcomes. Unlike pure software applications, CPSs sense, depend upon, and actuate physical phenomena which are not entirely under control of the software. CPS complexity is exacerbated by the need for CPSs to adapt to ecosystem changes if they are to remain functional over extended periods.

Under TEEE, we developed *dynamic profiling* tools and techniques to explore CPS performance envelopes. System maintainers can apply TEEE to analyze change events from the driving ecosystem, apply them to CPS designs and requirements, determine root causes of issues due to the changes, and identify design variants that provide the required functionality under the new circumstances. Dynamic profiling includes measurement instrumentation to extract information from models and from the CPS itself, cyber-physical stimuli to exercise the CPS, and methods to map the measurement results back into the system model. Our results enable identification and visualization of functional and resource limitations that impact the ability for the CPS to meet operating requirements within its changed environment.

To deliver this capability, Adventium launched its Curated Access to Model-based Engineering Tools (CAMET) library.[1] Adventium will use CAMET to transfer Adventium's Model-Based Engineering (MBE) technology, including TEEE stimulus synthesis, measurement synthesis, and models. The CAMET library includes MBE tools, documentation, models, and other materials to assist system developers and maintainers. We will distribute tools and updates to sponsors via the library. We will use sponsorship fees to support the library itself, while other projects will support new tool development and major tool updates.

[1] `http://www.camet-library.com`

2 Introduction

2.1 Relevance and Significance

CPSs can interact with the physical environment by sensing external state, transferring kinetic and potential energy, computing solutions to affect desired outcomes, and driving electrical, optical, and mechanical actuators to achieve those outcomes. Unlike pure software applications, CPSs sense, depend upon, and actuate physical phenomena, as shown in the control loop in Figure 1. The "process" in the figure is the real world. Even when CPSs are built around good models of the physical world, over long durations, the models might no longer track to unanticipated trends in environmental conditions, aging effects on mechanical systems, and component upgrades and modifications. The software in long-lifetime CPSs must adapt to these ecosystem changes if they are to remain functional over extended periods.

The CPSs, however, might not directly sense all aspects of their environments, especially those aspects of the environment which were not considered significant during original development. For example, a hydraulic fluid pump in an aircraft might not have a "pump wear sensor," because the original designers expected the unit to be retired before the pump exhibited significant wear.

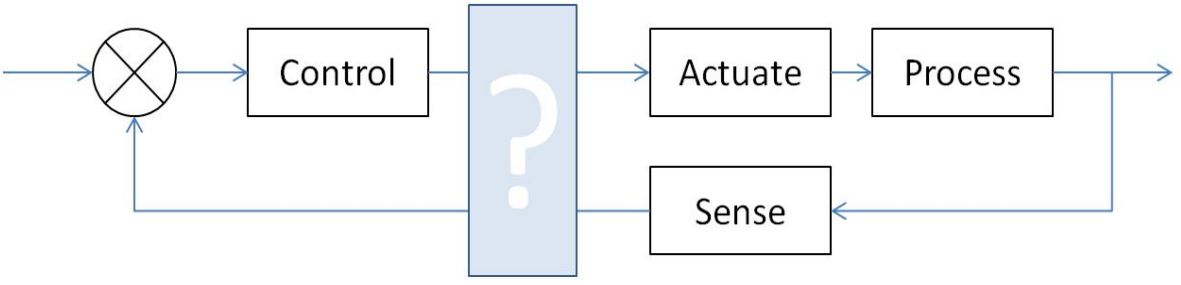

Figure 1: *This controller example shows where the path to the physical world can be a significant source of evolutionary change*

Over time, changes occur, such as field failures, component modifications, and the desire to use the CPS in new environments. These changes can lead to different outcomes:

1. Benign, where the CPS functions as desired and still meets its original design requirements. Examples include environmental conditions such as radio interference which is outside of the original tested and guaranteed operating conditions, but for which the system has sufficient sensing and capacity to continue to satisfy requirements.
2. Off-specification behavior, where the CPS no longer meets performance or functional requirements. For example, a network connection might change from a point-to-point connection to a multi-hop network with long latency and jitter. This might prevent the real-time control algorithms from converging. Another change could be new pump tubing introduced in the field with unexpected stiffness properties, resulting in undesired flow rates.
3. Latent defects, where the originally deployed system tests do not address use cases the designers did not envision, such as the need to connect pumps to external monitoring and predictive maintenance systems. Subsequent application of the system might produce valid use cases, but lead to identification of flawed requirements or design, or software that must be updated.

4. Use case changes that necessitate requirements modification. An example might be a CPS that is moved to an adjacent domain, e.g., pumping materials with significantly different viscosity than originally intended, or supporting file systems with drastically different behaviors, such as moving from local storage to Network Attached Storage (NAS), or a NAS moving from a dedicated device to a virtualized remote server.

TEEE is a user-in-the-loop tool-chain for measurement and stimulus synthesis. TEEE helps the system maintainer detect, diagnose, and identify ecosystem changes to target CPSs and the impact of those changes to target performance requirements. It helps the maintainer evaluate the ability of alternative configurations of the CPS to satisfy requirements in the context of the changes. As shown in Figure 2, external change events start the TEEE process. The figure shows numbered user actions that follow an iterative process to explore CPS behaviors as the environment changes.

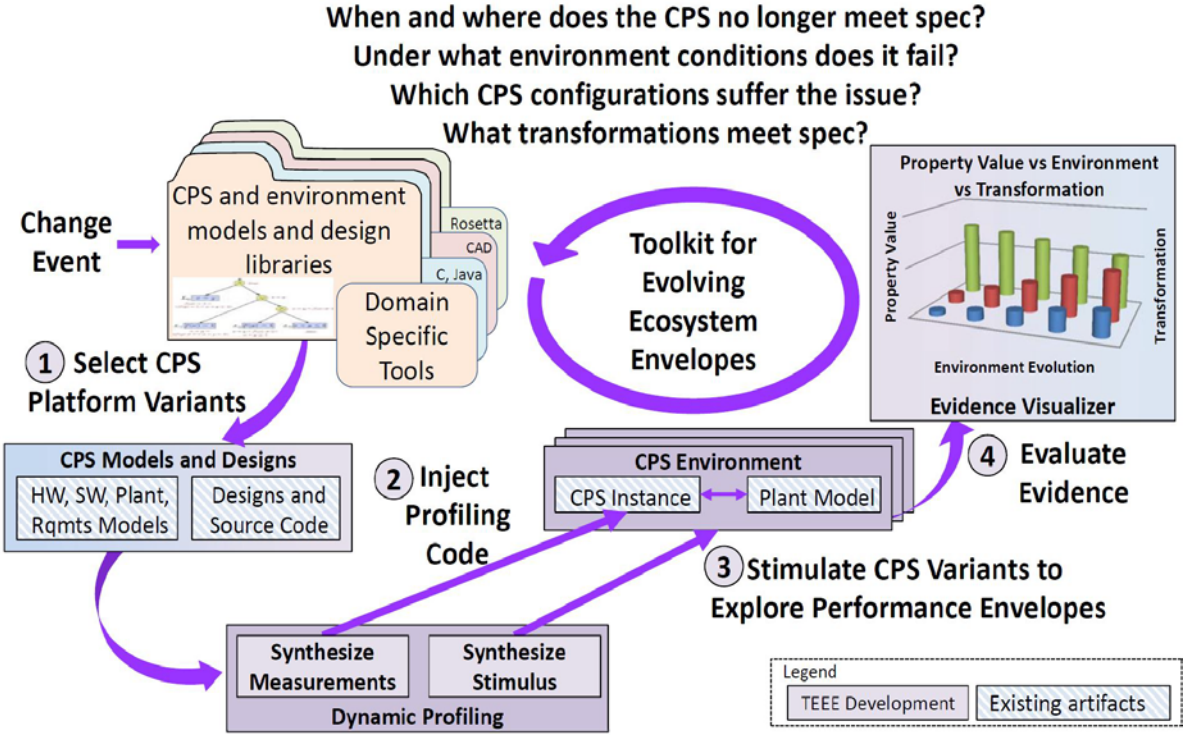

Figure 2: *Toolkit for Evolving Ecosystem Envelopes functional flow*

When an initiating change event occurs, the maintainer will start at step 1 by selecting from the design history file a version of the device implicated by the change event. An example event is a component failure in the field, requiring an update due to parts obsolescence. For new systems developed with TEEE, the design history file is developed along with the CPS, and includes integrated requirements, models, designs, and source code. For legacy systems, maintainers will need to create models of the target system.

At step 2, the maintainer will inject dynamic profiling code, using the TEEE Measurement Synthesis tools. This synthesis process, described in Section 3.3, reasons with the architecture models and the CPS source code to infer system behaviors [18, 17, 19], as well as generate and insert dynamic profiling code to produce, collect, and report the measurements. Measurements could include general measurements to reevaluate existing requirements if the implications of a change event are unknown. Alternatively, the maintainer will use TEEE to synthesize specific

measurements to diagnose observed anomalies, such as instrumenting the critical path for pump rate control if the pumping rate is not tracking properly.

Next at step 3, TEEE generates stimuli to evaluate CPS performance envelopes based on the change event and the desired measurements. CPS requirements and architecture models drive this synthesis process, described in Section 3.2. Similar to the measurement instrumentation, the stimuli drives exploration of the overall *operational envelopes*. Operational envelopes are regions in which the CPS is intended to correctly operate. For a fluid pump, an example envelope might include a space defined by flow rate, environmental temperature, and fluid viscosity. As a pump ages, however, the shape of the envelope can change. Under ideal conditions with convex state spaces, only the boundary conditions need to be evaluated. In the real world, the whole ecosystem must be considered. Software is only one component. The ecosystem includes environmental conditions that might not be directly sensed by the CPS, but which still impact the performance of the system. Therefore, TEEE synthesizes stimuli for environmental conditions both within and around this envelope.

In step 4, TEEE organizes the measurement results to help the designer identify issues and root causes. For example, if the CPS is not meeting the latency requirement for a command response, the measured latency through the components involved in the command response can be displayed and compared against the nominal or previous design.

2.2 Background and State of the Art

TEEE augments the DARPA BRASS program which is intended to create resilient systems that have robust and functional 100+ year software. BRASS has roots in autonomic computing [10] in which systems manage themselves given high-level objectives. TEEE monitors the system to determine error causes and possible adaptations, rather than the larger task of managing goals and objectives of the external system administrator. Part of ensuring resilient long lifetime software includes accounting for unanticipated uses of systems and environmental changes. TEEE uses dynamic profiling components to determine whether environmental changes and/or changes to the expected System Under Test (SUT) use cases are the cause of observed errors.

TEEE synthesizes dynamic measurements to capture physical world and other interactions that are not directly apparent from static analysis of the source code, and will include temporal effects and side effects in the CPS's environment. In contrast, the pure software world has the luxury of contract-based (or requirements-based) programming. Static analysis tools which work on simple sequential programs can prove properties about whether the code satisfies the functional characteristics defined in the contract. CPSs that interact with the physical world do not enjoy this luxury since they deal with embedded systems rich with side effects. Example side effects include writing to General Purpose Input/Output (GPIO) processor pins or memory-mapped Input / Output (IO) locations which causes changes to the state of something outside the software, such as actuator position or the addition of heat to a system. Static analysis of only the software is insufficient to determine how the pump will behave under these environmental conditions.

Since change events, such as those enumerated above, impact requirements, we drive TEEE by the CPS requirements. Typical design-for-test and unit-test approaches that start with software behavior will evaluate the target *software* against behavioral requirements. These methods only address a small fraction of issues, with the majority of defects actually arising from requirements [14].

TEEE leverages model-based development techniques for requirements, design, architecture, configuration, and automated measurement and stimulus to identify root causes of anomalies. In

contrast, state of the practice development processes in industry still largely rely on trial-and-error test-based software coding.

TEEE models have fixed, clear semantics from which the actual target CPS software is generated and configured. Our core language, Architecture Analysis and Design Language (AADL) [7, 25], is extensively applied in avionics, including within industry consortia such as the System Architecture Virtual Integration (SAVI) and Future Airborne Capability Environment (FACE) efforts. This provides TEEE with a ready base of users. Adventium has developed extensive AADL tooling and provided language extensions and tools to the DoD, which are being incorporated into the Army's Joint Multi-Role Technology Demonstrator (JMR-TD) efforts. In contrast, universal modeling paradigms suffer from imprecise semantics which often result in models used primarily for initial documentation, and are not maintained to reflect the actual system.

Rodriguez et. al. [23], model the security and specifically the resilience of systems in Universal Modeling Language (UML) models. Their analysis and modeling of security requirements exposes the underlying relationship between security and dependability. Similarly, TEEE uses the dynamic profiling components (Section 3.2 and Section 3.3) to uncover constraints in the system including security requirements. Rugina et. al. [24], present a framework for modeling dependability using the AADL [8], [7] and Generalized Stochastic Petri Nets (GSPNs). Their framework includes an AADL error model to present a full picture of the dependability for the user. Their framework is used to determine the reliability, availability, and safety prior to system deployment. TEEE extracts these requirements, including these dependability properties, stimulates the system, and evaluates performance against requirements to determine if they are satisfied in the event of an environment change or off-specification use when the system has been deployed.

Arafeen and Do [1] use requirements to prioritize test cases and more quickly determine faults. Their prioritization scheme clusters the requirements and prioritizes the cluster based on the priority of the requirements within. TEEE's test case prioritization scheme (Section 3.2) also uses system requirements to create and prioritize test cases. TEEE also takes into account whether the test case (and subsequently the requirement) has previously exposed an error. Dreossi et. al. [6] detect errors in machine learning components of CPS systems, such as in Lane Keeping Assist Systems in cars, by formulating it as a falsification problem for the model. TEEE similarly uses the model requirements to create test cases and determine errors within the CPS.

Adaption in systems (CPS or software) research is focused primarily on automatically creating patches for software. The GenProg system, Le Goues et. al., [12], uses genetic programming to automatically repair software defects given a set of test cases. The ClearView system [20] automatically patches errors in deployed software without access to source code or debugging info. ClearView learns normal execution, detects failures while monitoring execution, and generates a patch. While ClearView works on deployed systems, as TEEE does, it discovers errors by learning 'normal' execution and would be unable to discover error if the 'normal' execution changes (such as a system use case change). Converse to these software-only approaches, TEEE helps the maintainer find and repair issues stemming from the underlying architecture as well as software errors. TEEE models variant components in the CPS architecture and, when an issue arises, permits the maintainer to explore alternate architecture configurations. This exploration helps localize the root cause, as well as identify configurations where the issue does not manifest.

2.3 Organization of this Report

The major sections of this report are as follows:

- Section 3: The technical section documents the work performed on this project to develop stimulus and measurement synthesis tools.
- Section 4: Describes the results of the feasibility study.
- Section 5: Captures conclusions and next steps.

3 Methods, Assumptions, and Procedures

Resilient systems can be described as, “Trusted and effective out of the box in a wide range of contexts, easily adapted to many others through reconfiguration or replacement, with graceful and detectable degradation of function [16]." Resilient systems are also “expected to continuously provide trustworthy services despite changes in the environment or in the requirements they must comply with [26]." TEEE provides root cause analysis and error adaption to support CPS resiliency.

Figure 3 highlights the context of the TEEE approach, showing change events from the driving ecosystem applied to CPS designs and requirements, helping the maintainer evolve the CPS. Dynamic profiling includes measurement instrumentation to extract information from models and the CPS itself, cyber-physical stimuli to exercise the CPS, and methods to map the measurement results back into the system model. The result enables identification and visualization of functional and resource limitations that impact the ability for the CPS to meet operating requirements within its changed environment.

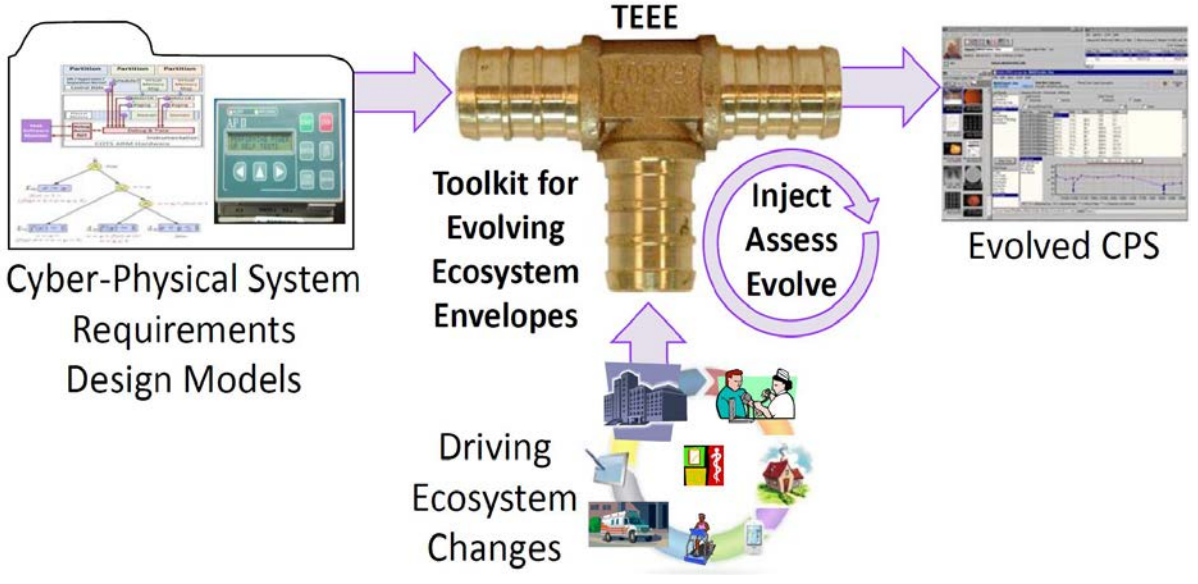

Figure 3: *TEEE helps direct CPS evolution based on inputs from the existing CPS and ecosystem changes*

The remainder of this section is organized by primary technical accomplishments, including modeling, stimulus synthesis, and measurement synthesis.

3.1 CPS Requirements Models to Drive Synthesis

TEEE performs its reasoning based on inputs from models of the SUT. We targeted a medical CPS, a Patient Controlled Analgesia (PCA) infusion pump, for demonstration purposes. Medical systems such as this share significant goals with weapon system CPSs, including safety, fault tolerance, and security. Research based on such a medical CPS, however, has the advantage of

few Internet Protocol (IP) restrictions. Hardware, software, requirements, and even models are publicly available, which makes it a useful research platform.

3.1.1 Goals

CPSs interact with the physical environment by sensing external state, transferring kinetic and potential energy, computing solutions to affect desired outcomes, and driving electrical, optical, and mechanical actuators to achieve those outcomes. The CPS, however, might not directly sense all aspects of its environment, especially those aspects of the environment which were not considered significant during original development.

For example our SUT is a PCA infusion pump which is rated for tubing with an inner diameter of 0.054". However, residents of less developed countries often use whatever equipment is available to them, often without standard safety procedures or support resources. These users may only have access to tubing with a 0.0033" inner diameter, which may speed pump degradation due to the increased resistance, and ultimately change the rate of flow of medication.

TEEE requires CPS models represented in a form that supports iteration over configurations representative of potential evolutionary scenarios. The models must also represent non-functional performance requirements characteristics and mission costs (e.g., latency, throughput, power). The system maintainer must be able to explore the ability for the system to satisfy system requirements for desired use-case scenarios.

3.1.2 Accomplishments

Our CPS includes the original system, and evolutionary variants including mechanical, hardware, and software variants. We developed a software-driven PCA infusion pump to serve as our demonstration CPS. The software includes a controller, display, and plant model executing on an ARM-based embedded prototyping card. This software is partitioned, both as Linux processes and Xen virtual machines, into safety-critical and non-safety-critical components.

This CPS design allows users to drive the actual mechanical infusion pump under software control. Users can specify infusion parameters on a interactive Graphical User Interface (GUI) display. The target software is hosted on an embedded prototyping board. Figure 4 shows our exemplar PCA pump functional software architecture, including multiple partitions, primary communications channels, and example output display captured from a functioning system. Example screenshots of the GUI are shown on the left of the figure. The GUI runs within a Xen partition on the prototyping board. Other partitions handle networking, the cyber physical abstraction layer, and the controller. A model of the software partitions and communication channels is shown on the right of the figure.

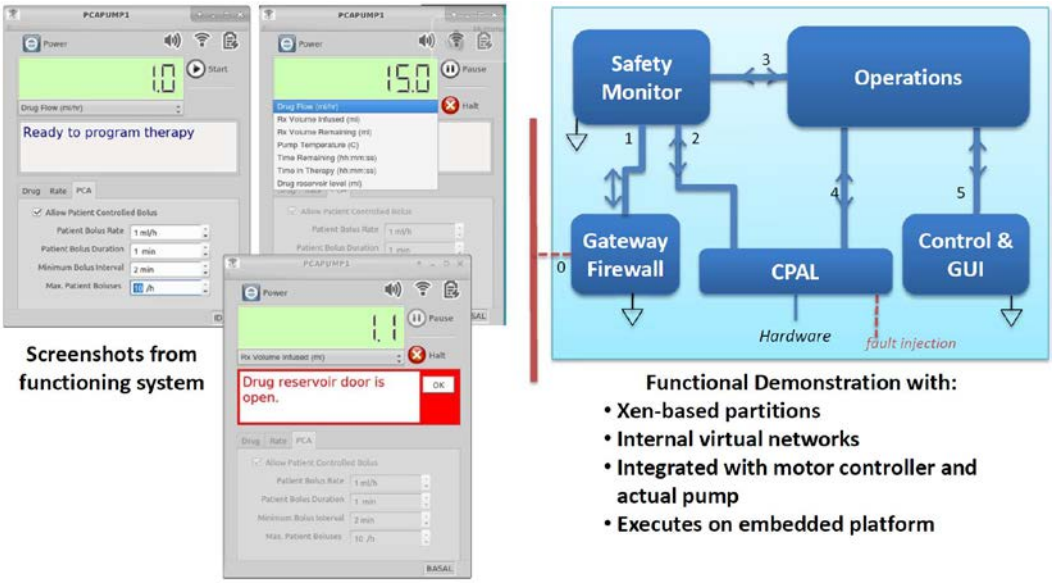

Figure 4: *Exemplar pump functional software*

Our PCA infusion pump hardware is a torn-down commercial pump, with a new modular embedded controller that we are using for prototyping and experimentation, as shown in Figure 5. We integrated switches that duplicate safety features on the real PCA pump, such as drug door open, drug vial present, and line occlusion. We placed Light Emitting Diodes (LEDs) across each stepper motor coil drive so we can visually perceive coil activation. The GUI in the figure is displayed remotely on the laptop to the left. It can also be displayed directly on an HDMI monitor. The embedded board runs a partitioned PCA pump and drives the motor controller. LEDs on the motor controller show coil energization. On the right is the PCA infusion pump driven by the motor controller. The red fluid in the tubing is food coloring in water pumped from the syringe in the cabinet. Changing the pumping rate specified in the GUI changes the speed with which the motor turns (or LEDs flash). The LEDs provide a useful method to visualize how the coils are successively energized to drive motor rotation.

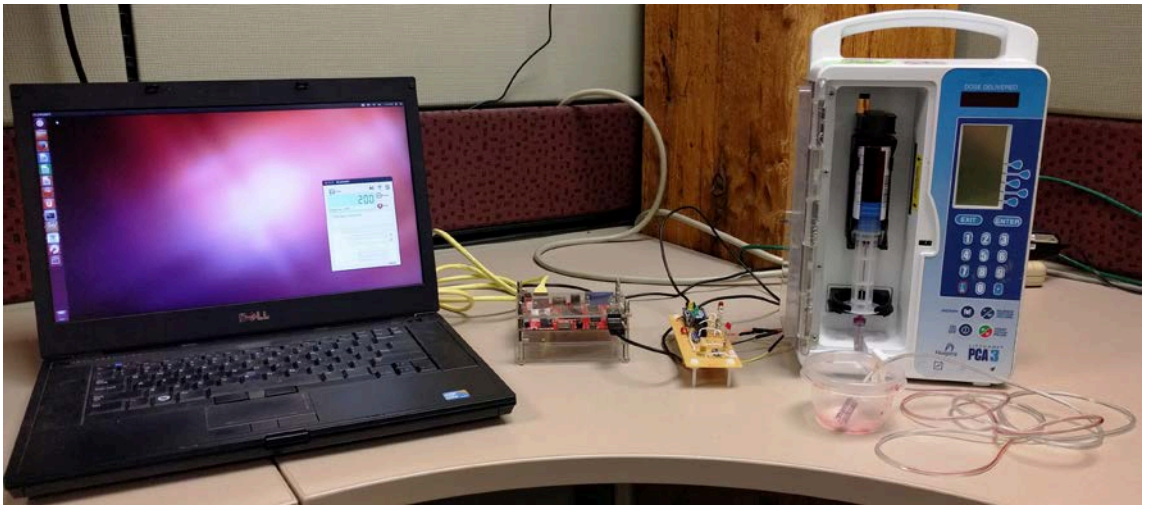

Figure 5: *Mechanical PCA infusion pump used as CPS for experimentation and demonstration*

Modeling

We modeled requirements and components of the SUT in AADL. AADL enables analysis of architectural features, so-called “non-functional” or performance properties, of a system and the relationships between the components. AADL supports modeling design variants coherently within a single AADL model, which facilitates both model maintenance and exploration of variants. AADL defines component types that include all externally visible features, separately from implementations, which model component internal behaviors. For example, this allows modeling multiple fault management approaches in a single model so they may be evaluated and compared. The AADL model also specifies the hardware, software, and binding of the software to the hardware. We modeled hardware and software variants in AADL, and can select specific configurations of those variants. We can realize select instances in hardware and software, and measure actual performance with the above mechanical infusion pump. This provided a research target for the stimulus and measurement synthesis research tasks.

Figure 6 shows the top-level AADL screenshot of the components and interconnection of the demonstration PCA infusion pump, including the power, interface logic, and motor controller. The fine text is not intended to be readable - the intent of the figure is to portray the high level components and interconnections.

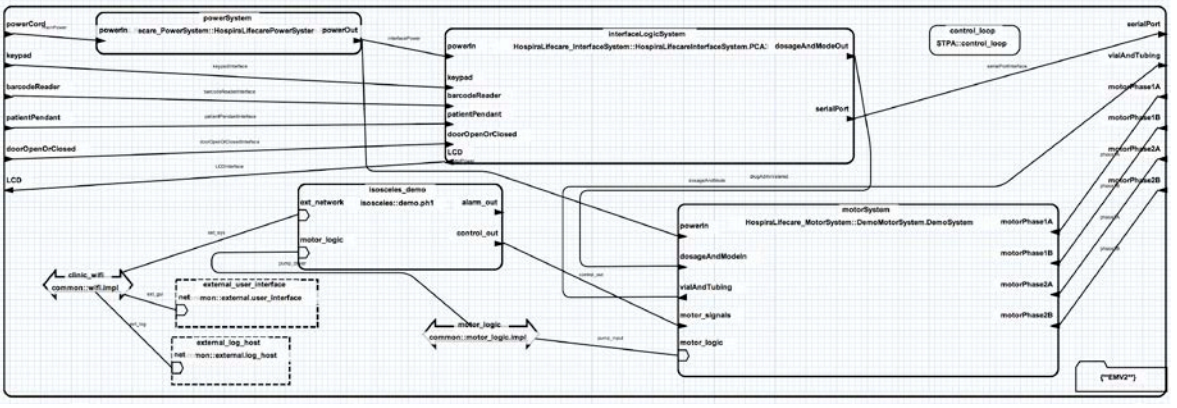

Figure 6: *Top-level AADL graph of the PCA infusion pump*

TEEE’s semantics include physical characteristics of the environment outside of the SUT. We captured those characteristics as properties in the AADL model as shown in Figure 7. The properties shown represent mode-specific failure rates of the pump’s motor controller. For the example shown, the failure rates are predicted. Once a CPS has been fielded, the maintainer can update the model with observed failure rates.

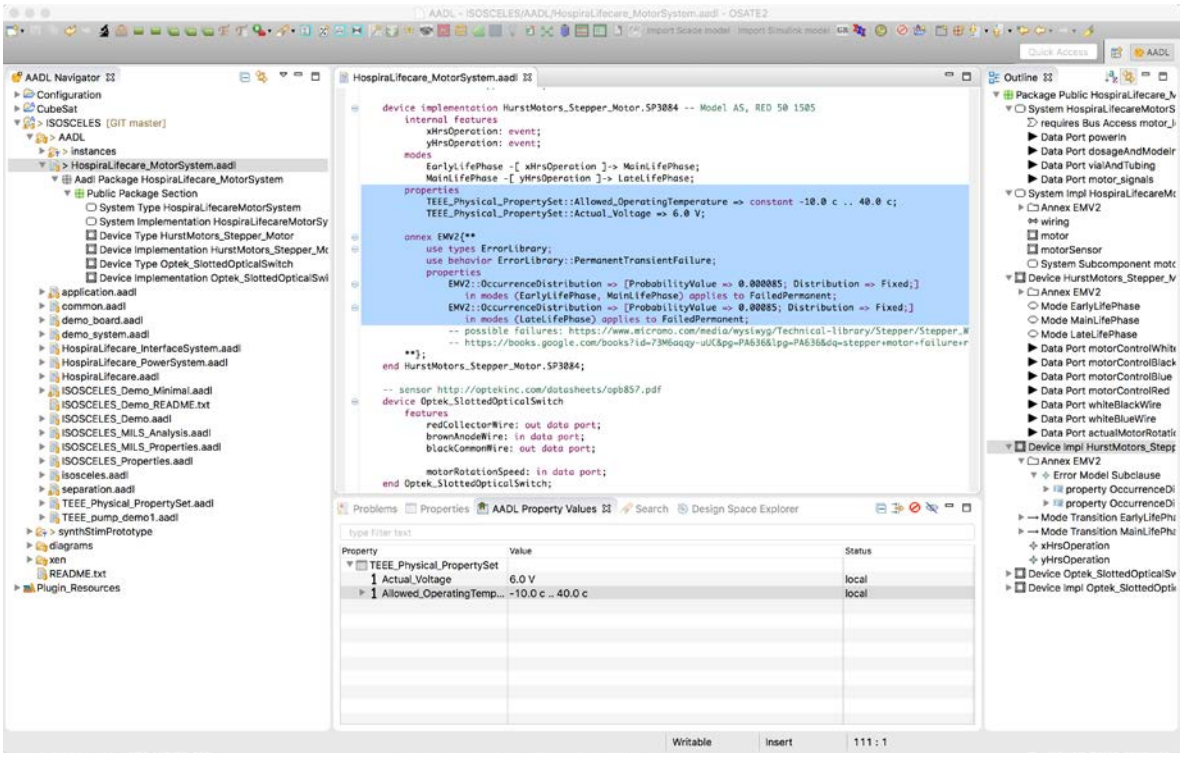

Figure 7: *OSATE Screenshot of AADL model properties*

Based on this relatively simple PCA infusion pump model, Figure 8 shows a graph of example tubing, medication, and motor-controller variants of the PCA pump. (The fine text is not intended to be readable - the intent of the graph is to portray the simple structure of the variant graph.) We can vary components of the system, such as the target processor architecture, and different boards which host that processor. We can also vary the power source, such as different batteries or line powered supplies. We developed a tool to extract the variant graph from our AADL model. Maintainers can make choices at each variant point and configure a specific instance of a PCA pump implementation and environment use case. This particular graph shows both hardware and environment variants. The environment variants, are shown at the bottom and include different tubing types and different drug media and temperatures. This small graph represents 104 different device variants. We experimented with a model of a CubeSat; just varying the available hardware and basic software options, and fault tolerance options resulted in 10^{22} variants.

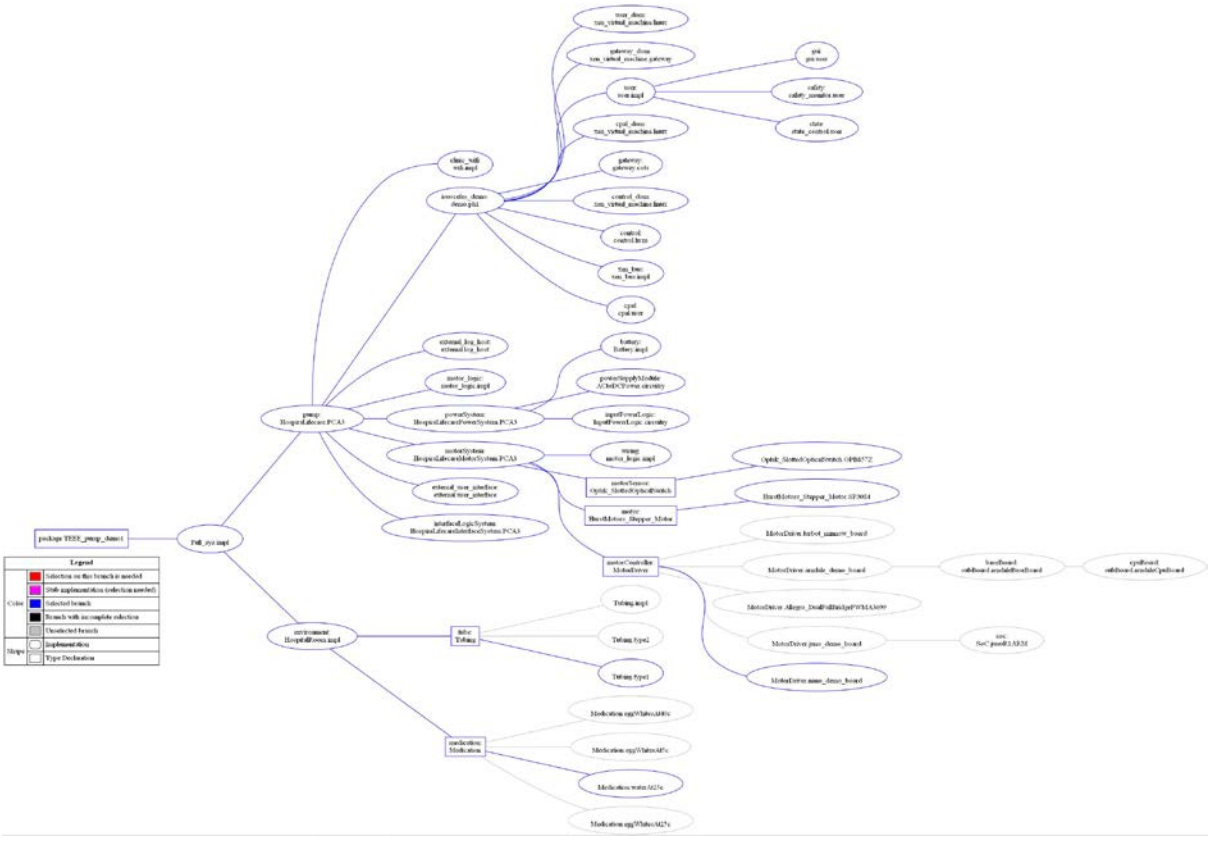

Figure 8: *Graph of example variants of the PCA pump (text labels not intended to be readable)*

Additional variants not shown in the graph include selection of a hypervisor or separation layer (e.g., Linux, seL4, or Xen), operating systems (e.g., none, MiniOS, Ubuntu, CentOS), communications and messaging protocols, application layers, and for our primary example, selection of software controller (e.g., open-loop, PID), and controller tuning parameters. The model includes physical properties, such as voltage levels of output pins used to drive the motor controller. This model bridges software performance to hardware voltage levels and mechanical pump movement, providing a comprehensive cyber-physical system view from software to hardware.

Altogether, this represents an enormous transformation space. For CPSs fielded for long durations, induced changes may be to the environment in which the CPS operates (e.g., moved to different elevations, or used to pump completely different fluids). The changes might also be to parts of the CPS itself, such as replacing worn obsolete components with new components. In any of these scenarios, the system maintainer may want to know, “Does the system still work? If not, what specifically isn’t working, and what is the root cause of that?” If there are choices, such as which new board to use to replace an obsolete board, the question might be, “Of my possible choices, which variant is most robust across my expected use cases?”

Component `implementations`, an instantiation of a component, may have subcomponents which themselves may be component types or implementations. A component type may have any number of implementations, all of which look identical from outside. By having multiple implementations for a component, different design alternatives can be modeled. This allows many design variants to be captured in a single model so they may be evaluated and compared. Over the lifetime of a long-lived CPS, maintainers may add alternate implementations and components to the system’s model as technology advances and parts become obsolete.

We use Adventium's Design Space Explorer (DSE) Open Source AADL Tool Environment (OSATE) plugin, shown in Figure 9, to select a specific instance of the test system to evaluate. For the PCA infusion pump, this includes selecting specific environment and system subcomponents (such as the tubing size and the motor driver we are interested in.

Figure 9: *OSATE Screenshot of Design Space Explorer tool showing specific modeling choices*

We then extract requirements properties from this instance to drive stimulus synthesis, which we address in the next section.

3.2 Stimulus Synthesis

The Stimulus Synthesize Algorithm (SSA) is the TEEE tool which probes the SUT operating envelope with a test case suite that it creates based on requirements extracted from the AADL model. For each component in the model, the SSA creates a test case that corresponds to each variable's allowable range and test range. To reduce the number of test cases and subsequently the time it takes to test the SUT, the SSA combines the test suite into pairwise test cases. SSA then prioritizes the test suite to increase the chance of finding failures early during the test sequence.

3.2.1 Goals

The goal of the Stimulus Synthesize Algorithm is to exercise potential CPS transformation variants to identify performance envelopes. To accomplish this, SSA provides the system maintainer with the following capabilities:

- Extract environment and CPS variant requirements from models.
- Automatically generate target CPS variant configurations.
- Prune configurations that can be statically eliminated.
- Automatically build target variants based on model specifications.
- Automatically generate test specifications.
- Apply combinatorial test techniques to prune tests based on redundancy analysis, coverage requirements, and boundary values.
- Evaluate non-functional performance constraints that drive software-physical interactions.
- Prioritize testing based on risk exposure of previously discovered errors.

3.2.2 Accomplishments

TEEE first extracts system requirements from the AADL instance model of the CPS system for the TEEE dynamic profiling components. Listing 1 shows an Extensible Markup Language (XML) snippet extracted from one implementation of the motor component. The specific motor modeled is called 'motor', its parents are defined under the `<Parents>` tag. The criticality of the component is defined by the user and annotated with the `<Criticality>` tag. Lastly the requirements of the component are defined using the `<Variable>` tag. Each variable may define an allowable and test range as well as the actual value. We include an optional test range specification, since the actual functioning range of a variable might be wider than the allowed range indicates. The requirement on the motor component defines the variable *Operating Temperature* as having an allowed range of $-10\,^{\circ}$Celsius(C) to $40\,^{\circ}C$.

```
<Component type="device" implementation="motor">
  <Parents>
    <SystemRef type="system" implementation="motorSystem"/>
    <SystemRef type="system" implementation="pump"/>
    <SystemRef type="system" implementation="Full_sys_inst"/>
  </Parents>
  <Criticality>0</Criticality>
  <Variable name="OperatingTemperature" units="c">
     <allowed>
    <real min="-10.0" max="40.0"/>
     </allowed>
  </Variable>
</Component>
```

Listing 1: An XML requirement on the motor component of the PCA pump that has been extracted from the AADL model.

Based on the extracted property list from the model instance, SSA will then synthesize and prioritize the analysis to perform on the SUT. Figure 10 is an example SSA screenshot that shows

test cases and specific patterns generated for environmental temperature testing of a motor controller.

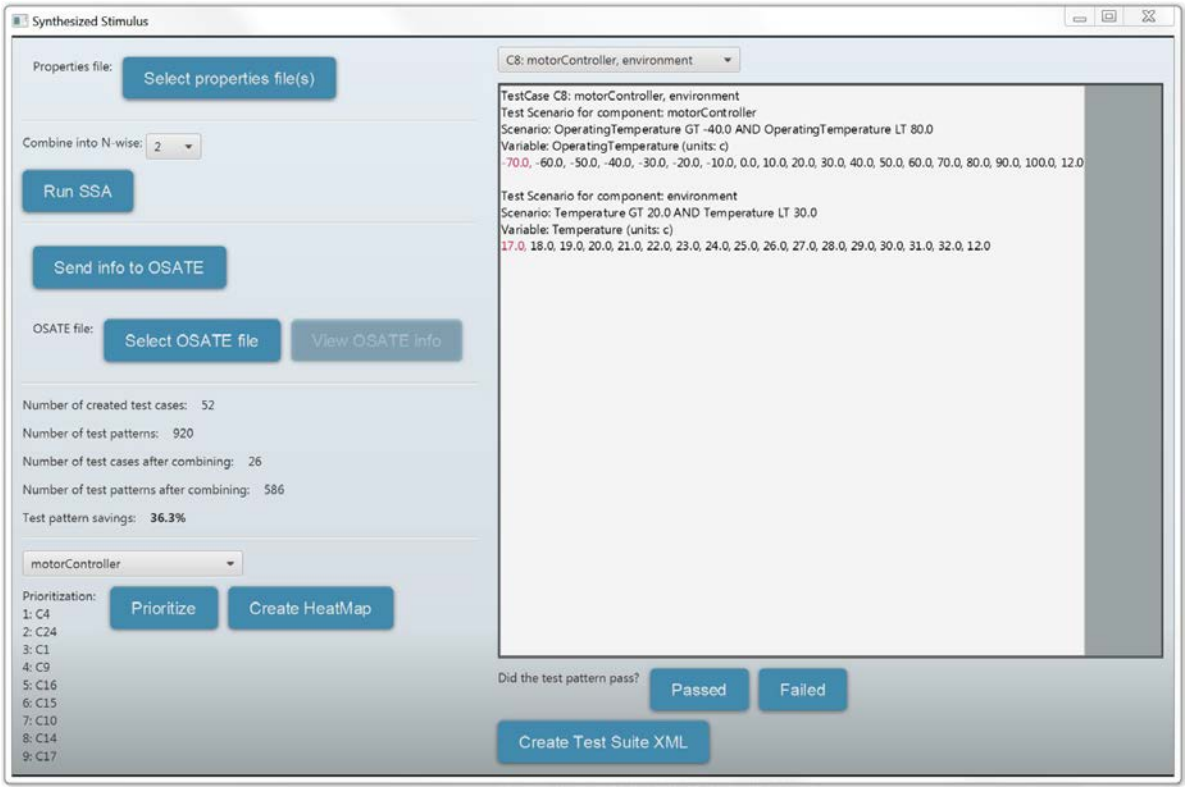

Figure 10: *SSA Screenshot*

For each component in the model, the SSA creates a test case that corresponds to each variable's allowable range and test range. The SSA is derived from earlier Rosetta-based test generations efforts [22]. We define a *test case* as a *test scenario*, a Boolean condition to be applied to a variable; and a *test vector*, a set of inputs to be substituted for the variable in the Boolean condition.

The system requirements for the motor component (earlier Listing 1) define one variable with an allowable range. Therefore, SSA will create one test case for that component. The test scenario is the Boolean condition: $-10^\circ C \leq temp \wedge temp \leq 40 C^\circ$. This will test if a particular component in the CPS, the motor, correctly performs under the temperature range for which it was designed. SSA creates the test vector for each test case using the step value specified in the requirement. If the model does not specify the step, SSA will automatically step based on the lowest non-zero decimal in the requirement specification (e.g., 200 has a step = 100, 0.34 has a step = 0.01).

SSA will create a test vector for the operating temperature variable by enumerating each value between $-10^\circ C$ and $40^\circ C$ with a step of $10^\circ C$, or $(-10^\circ C, 0 ... 30, 40^\circ C)$.

Since boundary values have been implicated in faults within the SUT [15], SSA applies an additional n, where n=2 in the current prototype, vector values to each boundary.

Combine Tests

To reduce the number of test cases and subsequently the time it takes to test the SUT, the SSA combines test cases using the method by Lott et. al. [13]. The combination algorithm is a greedy algorithm [4], which randomly pairs test cases until there are none (or only one) left to combine. The SSA does not pair test cases which test the same variable (e.g., temperature) in the

test scenario. The example in Figure 10 shows a 36% reduction in total test patterns for a pairwise combination.

We found, as Lott et. al. did, that a higher order combination yields greater test pattern savings. The SSA defaults to pair-wise combination to reduce the risk of combining differently named variables which are actually the same (e.g., “operating temperature” and “temperature”). With pairwise combination, assuming independence, growth of the test space increases $log_2(x)$ where x is the number of independent requirements. Increasing the order of combination of test cases reduces the growth rate to $log_n(x)$.

SSA will combine these tests when the maintainer selects ’Run SSA.’ It then presents the N-wise combined test suite. The GUI also reports the number of test cases and patterns before and after combination.

Prioritize Tests

Based on the Fault-Recorded Test Prioritization (FRTP) technique [21], SSA next prioritizes the test suite with the intent to identifying faults early in the testing sequence. The maintainer may use the pull-down menu in the lower-left of Figure 10 to request prioritization of a specific component. In the upper right of that figure, the maintainer can also select a specific test case. In the example shown, the test case is varying motor controller operating temperature versus the allowed external environment temperature. After running the test, the maintainer can indicate success or failure for manually executed tests. In the example shown, the test successfully passes if the correct volume of fluid is pumped.

SSA uses each test case result to reprioritize the remaining test cases. Failures indicate a fault has been found in the component(s) being tested within the test case. FRTP iteratively extracts information from the testing process and does not need to be bootstrapped with information from prior test executions. FRTP prioritizes test cases based on previously found faults. Some components might be deemed *more important* than others. For example, if the PCA pump motor fails stopped, then the PCA pump will not pump fluid. If instead a sensor on the motor fails then the PCA pump may still pump some fluid, although not necessarily the correct amount. To encode this we added the criticality of the component to the prioritization algorithm based on an equation derived from the *Risk Exposure* metric [3].

$$RiskExposure(TS) = \frac{\sum_{tc \in TC} P(f) * C(f)}{|f|} \quad (1)$$

Chen et. al., defines the risk exposure metric (Eq. 1) as the probability of failure ($P(f)$) of a component in the current test case tc multiplied by the cost of failure of the components in the current test case ($C(f)$) and then divided by the total number of components in the current test case. In place of determining the probability of failure for each component in the test case we redefined $P(f)$ in TEEE to represent the number of times the components in the current test case previously failed any test case. Equation 2 shows the TEEE definition of $P(f)$ which is a novel extension of the Chen Risk Exposure metric. In TEEE, $P(f)$ is the sum of the identified faults for each component in the current test case over the entire test suite (denoted by TS). The cost of failure ($C(f)$), or criticality of a component, is annotated by the user in the AADL model (`<Criticality>` tag). The default criticality is zero, which means not critical.

$$P(f) = \sum_{tc \in TS} \left(\sum_{c \in tc} FDN(c) \right) \quad (2)$$

We also incorporated a test selection and prioritization algorithm based on Multi-Objective Evolutionary Algorithmss (MOEAs). We implemented and evaluated a MOEA against our current SSA test selection and prioritization algorithm. We used a popular off-the-shelf evolutionary algorithm suite, MOEAFramework, and recreated MOEA according to the paper Multi-Objective Black-Box Test Case Selection for System Testing [11]. We implemented the MOEA to maximize the failure probability of the selected tests, maximize the cost of failure of the components within the selected tests, and maximize the failure of probability of the components within the selected tests. To evaluate the MOEA against the earlier test selection and prioritization methods, we created four scenarios of the system under test, the PCA pump, with real world data: desert temperatures using the tubing with the correct size diameter for the PCA pump according to its specifications; desert temperatures with tubing with a smaller inner diameter than required in the specification; Minneapolis room temperature with tubing of the correct size inner diameter; and Minneapolis room temperature with tubing with a smaller diameter than required.

For this scenario, MOEA selected a set of test cases to run that had a higher overall cost of failure within the components tested than the SSA described above. MOEA also selected a set of test cases with components that had a higher failure probability than the SSA algorithm. However, the F-Measure of the selected tests, which measures the precision (how many selected tests are relevant i.e., find failures) and recall (how many relevant tests are selected), was higher in the earlier SSA algorithm, which is more desirable since it could decrease overall testing time.

Finally Grindal et. al. [9] looked at the effectiveness of test case combination and found better results when pair-wise test cases are combined with a single variable test strategy. As a final step, SSA therefore randomly adds *k* test cases to the test suite from the pre-combined list of test cases for the SUT. We choose a random *k* between 25% and 75% of the test suite size to test the prototype.

Once SSA generates the stimulus, the maintainer can apply that stimulus to the AADL instance, and run automated analysis tools, such as temporal analysis, to verify that performance requirements are satisfied. The maintainer can also manually apply this stimulus to the physical system and measure its performance, such as actual volume of fluid pumped versus the predicted volume.

Figure 11 shows example results from the automated analysis. This particular stimulus probed the motor controller with a specific temperature and voltage. The results shown here include the Reliability Block Diagram (RBD) analysis which calculates the mode-specific failure probabilities of components. It also checks that both the temperature and voltage were within the allowable ranges specified by the model instance. In this case all the tests passed.

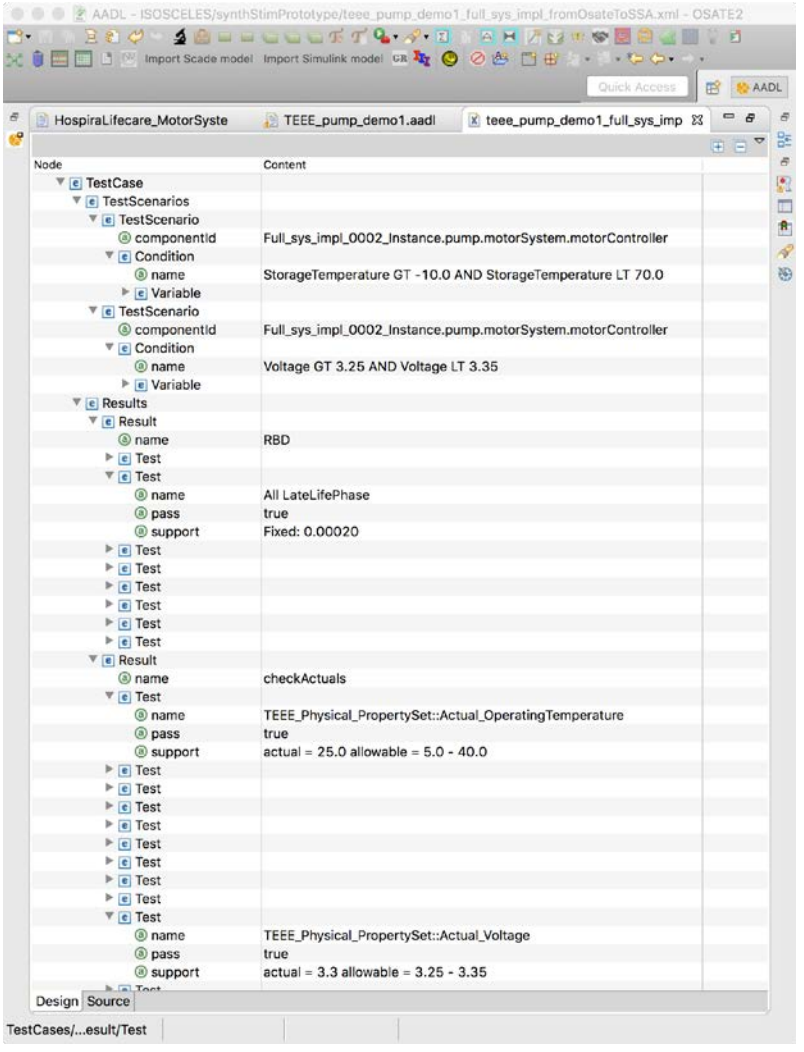

Figure 11: *OSATE Screenshot, showing automated test results*

Figure 12 is another screenshot of the SSA in action. The requirements file for the PCA infusion pump has been loaded and SSA has been run. The left side of the GUI shows statistics on the number of test cases created and the number of *test patterns* (the test scenarios from the test case and one test vector value from each test case) before and after combination. In this particular example, combining the test cases into pair-wise test cases creates a test pattern savings of 15% for this particular variant. The right side of the GUI shows a pairwise test case varying current and voltage. The report on the right shows that not all tests passed. In this case, under-powering the device lowered the reliability of the device to unacceptable levels. The Heat-map in the center of the display shows that the first test case (C0) exhibited the most failures compared to the other test cases. The issues identified in these tests are related to reliability. For a safety-critical device, a more appropriate response might be to trade availability for integrity, and transition to a safe state on low power. The maintainer might therefore decide to use a variant to adjust setpoints to implement this response.

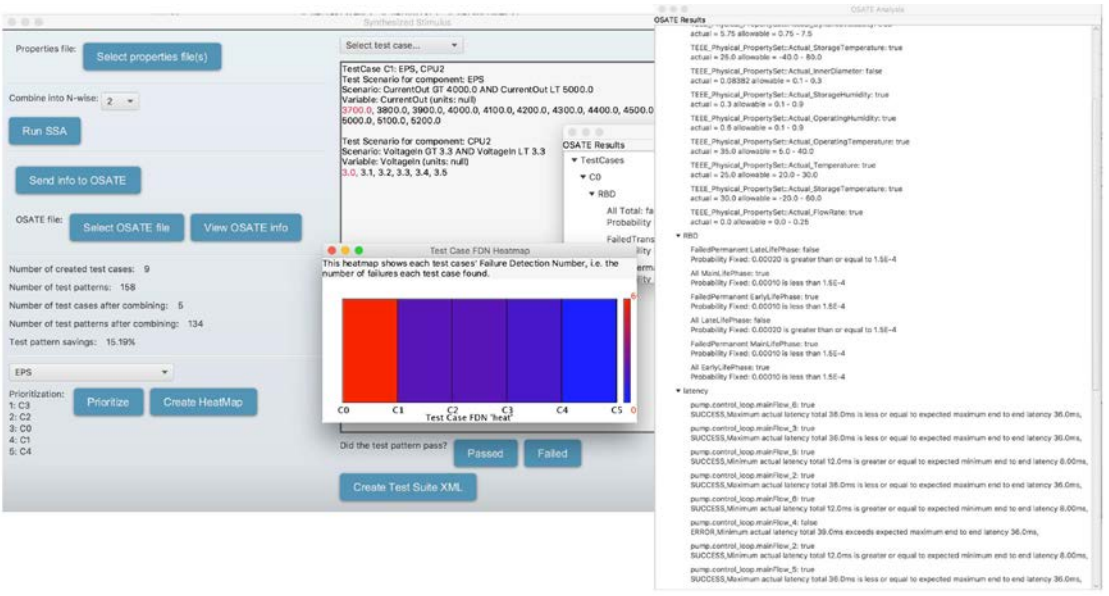

Figure 12: *SSA screenshot*

3.3 Measurement Synthesis

3.3.1 Goals

The goal of the measurement task was to *develop a means of gathering measurements from a target system in a verifiable way*. Some information needed to understand CPS behavior is not directly available through sensing. Designers may not anticipate long-term system behavior or may deem sensors too expensive. Thus, we use existing sensor data and information from the user to calculate values necessary for appraisal and adaptation.

The measurement approach is based on semantic remote attestation [5] where an appraiser must gather information from a system in order to determine trust. Here we use attestation to gather information from sensors over time and calculate needed run-time measurements. These measurements are then used to establish baseline and adapt the operational system.

To assure correctness of calculated measurements, we synthesize measurement protocols from formal specifications written in an Domain Specific Embedded Language (DSEL) defined in the Coq [2] verification environment. The DSEL formally defines the measurement protocol, capturing measurement acts, calculations, user inputs and units associated with quantities.

The synthesis approach de-sugars the DSEL representation by removing units and walks the new unit-less specification to synthesize C with calls to a standard, re-targetable measurement Application Programming Interface (API). The resulting C is then integrated into executable models and the actual CPS.

3.3.2 Accomplishments

3.3.2.1 Domain Specific Measurement Language

We designed and implemented a DSEL to represent measurement protocols. The language allows users to specify measurement acts sequenced with calculations for derived values. Our working example calculates the flow rate of liquid from an infusion pump:

$$flowRate := metersPerRevolution * motorSpeed * (\pi * (tubeDiameter/2)^2)$$

The DSEL representation for this calculation has the following form:

```
Definition flowRateCalc : Program :=
  (Measure MotorSpeed varMotor1)
   >> (Delay (Nats.natToNum 3))
   >> (Measure MotorSpeed varMotor2 )
   (* motorAverage = (varMotor2 + varMotor1) / 2 *)
   >> (Calc (vString (measurementToTYPE MotorSpeed)
             ((Void \ second)\Void) "motorAverage")
            (((TcalcValue (numTermVar varMotor2))
             +C+ (TcalcValue (numTermVar varMotor1)))
             /C/ (TcalcValue TWO)))
   >> (Measure MetersPerRevolution varMetersperRev)
   (* radius = vialDiameter/2  *)
   >> (Calc radius ((TcalcValue (numTermVar varVialDiameter))
                    /C/ (TcalcValue TWO)))
   (* area = radius * radius * PI  *)
   >> (Calc varArea
        (((TcalcValue (numTermVar radius))
         *C* (TcalcValue (numTermVar radius)))
         *C* tPI))
   (* answer =   metersPerRev * area * motorAverage *)
   >> (Calc (vString NUMERIC VolumetricFlow "answer")
            (((TcalcValue (numTermVar varMetersperRev))
              *C* (TcalcValue (numTermVar varArea)))
              *C* (TcalcValue  (numTermVar varMotorAve))))
   >> End.
```

Listing 2: DSEL flow rate of liquid calculation.

`Calc` terms represent calculations over measured values and other calculation results. `Measure` terms represent measurements performed on the target device as well as binding variables to values. `Delay` terms represent waiting between actions for a predetermined time. Terms of the form `*C*` represent calculations as indicated by the `*` symbols which can be multiplication, division, addition, subtraction and others as needed. Terms beginning with `var` indicate units such as area, meters per revolution, and motor averages.

3.3.2.2 Static unit checking

The DSEL language supports specification of units for measurements and performs unit checking as a sanity-check prior to synthesis. Unit checking is implemented using Coq's dependent type system.

When called on the above example, unit checking makes certain that unit compositions over operations result in the anticipated unit type. For example, when the unit checker looks at the calculation of tube cross section, it uses the definition of area (πr^2) and type units of quantities in calculations (radius and number) with multiplication to assure the definition does in fact calculate an area value.

3.3.2.3 Synthesis to the model and device

Following unit checking and verification the DSEL is synthesized to standard C code with calls to a customizable API for invoking measurements. Synthesis starts by erasing units from the

protocol specification resulting in a unit-less specification suitable for synthesis. The unit-less representation for the above example has the form:

```
  Utchain
 (UTmeasure MotorSpeed (idName NUMERIC "motor1"))
 (UTchain (UTdelay (Nats.natToNum 3))
 (UTchain (UTmeasure MotorSpeed (idName NUMERIC "motor2"))
 (UTchain
   (UTcalc (idName NUMERIC "motorAverage")
     (UTcalcDiv
     (UTcalcAdd (UTcalcValue (numTermVarND (idName NUMERIC
"motor2")))
       (UTcalcValue (numTermVarND (idName NUMERIC "motor1"))))
       (UTcalcValue (numTermValueND (Nats.natToNum 2))))))
 (UTchain (UTmeasure MetersPerRevolution (idName NUMERIC
"metersPerRev"))
 (UTchain (UTmeasure VialDiameter (idName NUMERIC
"vialDiameter"))
 (Utchain (UTcalc (idName NUMERIC "radius")
   (UTcalcDiv (UTcalcValue (numTermVarND (idName NUMERIC
"vialDiameter")))
              (UTcalcValue (numTermValueND (Nats.natToNum 2)))))
 (Utchain
   (UTcalc (idName NUMERIC "area")
     (UTcalcMult
       (UTcalcMult (UTcalcValue (numTermVarND (idName NUMERIC
"radius")))
                   (UTcalcValue (numTermVarND (idName NUMERIC
"radius"))))
       (UTcalcValue (numTermValueND Nats.PI)))
       ... ) : UTprogram
```

Listing 3: Unit-less specification suitable for synthesis.

This form is for use as synthesis input only and should not be written or read by the system developer.

The synthesis system walks the tree specified by the unit-less specification and generates C that includes API calls to a customizable measurer. The C resulting from synthesis of the above example has the form:

```
extern double __Measure(const char* identifier);

double getFlowRate() {
    double motor1;
    motor1 = __Measure("__MotorSpeed");
    sleep(3);
    double motor2;
    motor2 = __Measure("__MotorSpeed");
    double motorAverage;
    motorAverage = ((motor2+motor1)/2);
```

```
    double metersPerRev;
    metersPerRev = __Measure("__MetersPerRevolution");
    double vialDiameter;
    vialDiameter = __Measure("__VialDiameter");
    double radius;
    radius = (vialDiameter/2);
    double area;
    area = ((radius*radius)*3);
    double answer;
    answer = ((metersPerRev*area)*motorAverage);
    return (answer);
}
```

Listing 4: Synthesized C code.

The generated C maps calculation operations to standard C operations and variables to C variables. Calls to `__Measure` invoke measurement routines through the measurement API. The argument to `__Measure` names the measurement to be performed or a constant value input during execution. For example, `__VialDiameter` is the diameter of the medicine vial while `__Motorspeed` is the value of the motor's current revolutions per second.

Measurements are defined in the measurement API implemented using a `switch` to select from available measurements:

```
  double __Measure(const char *identName) {
    double value = -1;
    switch (nameToId(identName)) {
      case  VIAL_DIAMETER :
        value = .005;
        break;
      ...
      case  MOTOR_SPEED :
        value = getCurrentMotorSpeed();
        break;
      default :
        printf ("Unrecognized identifier '%s' argument to
__Measure\n",
                  identName);
        value = 0.0;
        exit (-1);
    }
    return (value);
}
```

Listing 5: Measurement definitions.

The `switch` multiplexes among multiple measurements and can be updated without modifying measurement protocol code.

3.3.2.4 Results

We successfully performed unit tests on our example calculations using the Coq verifier. Unit checking requires users to have knowledge of Coq. If we are to field this system broadly, we will need to provide more automation for unit checking.

We successfully constructed API interfaces for numerous system measurements. This requires developing an infrastructure that watches for measurement invocation and manages communication with the measurement code. The complexity of this interface is highly dependent on what is being measured. For example, reading constants is trivial. Gathering Revolutions per Minute (RPM) information, which is a series of measurements over time, requires setting up and tearing down a communication channel between measurer and target.

The C synthesized from the DSEL has been successfully integrated with both the CPS model and the system itself without modification. Our examples are limited in scope due to time constraints, but the C generated was successfully run on the model and the CPS example.

4 Results and Discussion

This TEEE seedling developed the ability to model and reason about software design strategies in the context of system-level environment evolution, as part of a system design life-cycle. This section discusses the results of applying these tools to a PCA infusion pump, showing the output of TEEE components, and demonstration that TEEE helps determine possible root causes.

Medical CPSs developed for first-world countries are often retired to developing countries after their service life expires in the first-world countries. In developing countries, however, resources are not always available to run these systems in the environment for which they were originally designed. We examined two scenarios which came from real world observations of PCA pumps used in developing countries.

Our first scenario is based on brown-outs, or low-power fluctuations in the external power supplied to the device. Brown-outs can cause the motor to run slower and subsequently the volume of medication pumped is less. To continue regular functionality, many medical devices contain a battery. However, very few working batteries make it to the developing world. Brown-outs therefore can have a significant effect on the device. First-world developers often do not consider extended brown-outs to be a primary use case, since we can usually rely on our national grids.

For example, one of our team members worked in developing countries to refurbish medical equipment. He once had to work with donated external defibrillators. None of these devices had working batteries. While the defibrillators were designed to function without a battery (slightly slower charge build up), they were clearly never intended to be used this way, since one of steps in the daily self test *required* the presence of a battery despite the battery itself not being present in the test.

For our PCA infusion pump, the model we selected does not have the means to directly sense the flow rates of the material that it is pumping. If a brown out occurs, but at a level that did not trigger a transition to a safe state (e.g. turned off), it might not be able to correctly determine how much material was pumped.

Our second scenario is based on the viscosity of the material being pumped. Untrained or overworked users may put the wrong medication into the pump. While there is a bar code reader on the PCA pump, it is easily bypassed. In developing areas, the original medications might not be available, or be too expensive, so they might be substituted with other materials. Additionally, temperature has an affect on the viscosity of liquids. Plasma is commonly used to treat patients with shock. If, for example, the PCA pump is used in an area which is very hot and without air

conditioning the medication could be *more* viscous. TEEE is able to determine the root cause of this error is the viscosity of the medication in the pump.

Egg whites are similar in viscosity to blood plasma, so we used that for our experiments. We ran experiments to confirm the validity of these scenarios. Water and egg whites were run through the PCA pump for 5 minutes and we varied the speed of the motor (100Hz, 50Hz, and 25Hz steps). We ran 5 experiments for each variant. The experiments showed a significant difference, using a paired t Test with $p < 0.005$, between volume expelled per step of the motor between 100Hz and 25Hz as well as 50Hz and 25Hz when using egg whites. Water showed a significant difference between 100Hz and 50Hz as well as 100Hz and 25Hz. The t Test resulted in a value of $p = 0.007$ when comparing 50Hz and 25Hz using water. The results of this experiment can be seen in Figure 13. The test results indicated a significant difference in volume expelled per step for water versus egg whites at 100Hz and 25Hz ($p < 0.005$). These experiments confirm the validity of the brownout scenario by showing the rate of the motor affects the amount of material dispensed. It should not, since each motor step should be a fixed amount. They also confirm the validity of the viscosity scenario showing materials at different viscosities affect the amount of material dispensed.

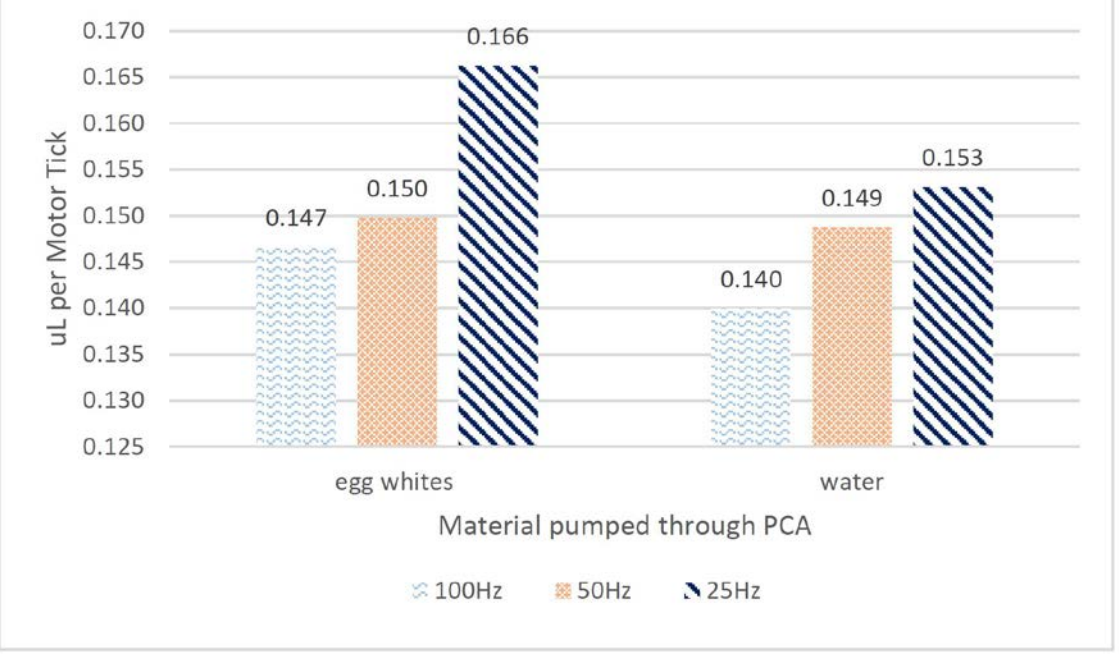

Figure 13: *Comparison of the volume of material pumped for different scenarios*

4.0.3 Confirmation of adaptation

The data from the PCA pump experiments show that there is a difference in the amount of material expelled when using materials of difference viscosities. Viscosity of the medication in the PCA pump, however, is a change within the environment that cannot be known via its sensors. To confirm that TEEE is able to determine the root cause of this scenario (material is of a different viscosity than is expected) and adapt to such changes we will dive into the output of each component. The first step is to model the PCA pump. One portion of that model is shown in Listing 6.

```
<Component type="device" implementation="tube">
 <Variable name="FlowRate" units="ulps" varType="real">
```

```
    <allowed>    <real    min="0.141"    max="0.147"/>
</allowed>
 </Variable>
<\Component>

<Component type="device" implementation="medication">
 <Variable      name="DynamicViscosity"      units="cP"
varType="real">
   <allowed> <real min="1" max="1.5"/> </allowed>
 </Variable>
<\Componen>
```

Listing 6: An extracted property snippet showing requirements in the viscosity scenario.

The tube component of the model includes a requirement that the flow rate of the medication must be between 0.141 and 0.147μLiters per second (μ*lps*) and viscosity of the medication must be between 1 and 1.5 Centipoise (cP). The SSA created 20 test cases from the requirements within the model which enumerated 3529 test patterns (the test scenarios of the test case and one value from each of the test vectors). After the SSA pair-wise combination step is run the test case suite size is reduced to 10 cases and 2674 test patterns, yielding a test pattern savings of 24%. The test cases in Table 1 correspond to these requirements.

Table 1: Test cases created for flow rate and medication viscosity requirements

Component	Test Scenario	Test Vector	Actual Value
Tube	0.141<*FlowRate*(μ*lps*)<0.147	0.139, 0.140, 0.141, 0.142, 0.143, 0.144, 0.145, 0.146, 0.147, 0.148, 0.149, 0.166	0.166
Medication	1.0<*Viscosity*(*cP*)<1.5	0.8, 0.9, 0.94, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7	0.94

A randomized user was simulated testing the PCA pump, i.e., running through the test cases and marking them passed or failed. Each test pattern had a 50% chance to mark its parent test case as failed, except the test case for the Tube component, shown in Table 1, which was marked failed each time. The test case suite was then prioritized on the tube component. The resulting prioritization along with detected failures (FDN) and risk exposure score is shown in Table 2.

Table 2: Test case prioritization for the tube component

Case Id	Component A	Component B	FDN	Risk Exposure
C5	tube	medication	2070	.60
C2	tube	power system	144	.35
C8	interface logic system	tube	44	.17
C1	motor	power system	114	.09

C7	pump	power system	120	.08
C9	environment	power system	110	.06
C4	pump	pump	12	.04
C0	pump	interface logic system	46	.02
C3	motor sensor	motor controller	18	.01
C6	motor controller	motor sensor	30	.01

Next, the information on the test case failures was sent to the Dynamic Measurement component to provide more information concerning the cause of the error. The Dynamic Measurement component models the calculation of mass flow rate as described previously. Working from measured values back to flow rate provides an alternative perspective on the failure. The mass flow rate equation is defined using Coq and verified using units analysis and using an execution semantics for the protocol description. Using information from testing and measurement, the user is able to determine the failure is likely that the medication is the incorrect viscosity rather than the alternative of improper tube diameter.

We also integrated the synthesized measurements into the PCA infusion pump motor rate controller. This controller is a Proportional, Integral, Derivative (PID) controller, and it runs in both the software-based functional simulation environment and on actual hardware. An example of results from the simulation is shown in Figure 14, overlaid on performance measurements from the actual hardware. Both environments show the rate controller bringing overshoot down to the desired rate. The PID controller relies upon the automatically synthesized dynamic measurer. The behavior from the two environments track closely, which means that the simulation environment can be effectively used for system development and analysis.

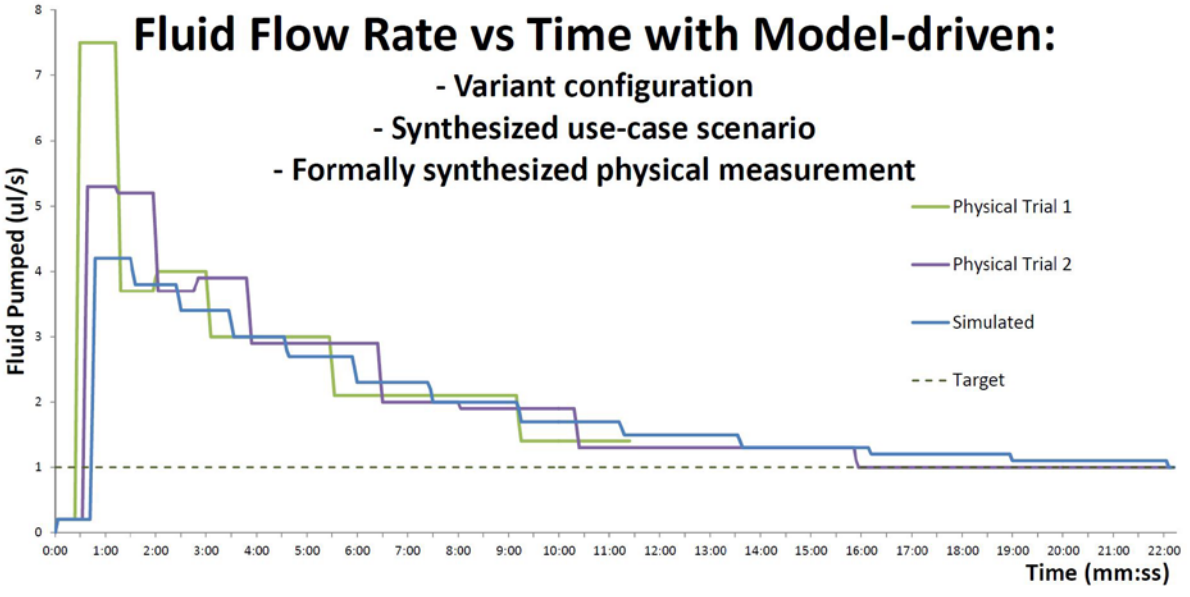

Figure 14: *Traces from simulated and real environments of PID controller performance*

We also experimented with a complex cube satellite model. The CubeSat model was developed under separate research into fault management strategies. We updated that model to include information about the external environmental conditions under which it was intended to operate. We were able to make SSA work on this model without requiring changes to the model structure itself, other than the additional properties. This is an important result, since it shows that

models developed for one purpose can be annotated with environmental information and analyzed by TEEE.

5 Conclusions

This seedling showed that TEEE helps address challenges in CPSs due to changing environment or use over time. We presented a real world example of environmental changes affecting the use of a decommissioned PCA infusion pump. The scenario was verified by a series of experiments performed on the model and the actual PCA pump. We showed the prototype is able to determine the root cause of the issue in the scenario using the SSA and Dynamic Measurements algorithms.

To deliver this capability to potential users, Adventium hosted these tools on its CAMET library. Adventium will use CAMET to transfer Adventium's MBE technology, including TEEE stimulus synthesis, measurement synthesis, and models. The CAMET library includes MBE tools, documentation, models, and other materials to assist system developers and maintainers. We will distribute tools and updates to sponsors via the library. We will use sponsorship fees to support the library itself, while other projects will support new tool development and major tool updates.

As TEEE is matured and deployed, Adventium will generate revenue to cover our Non-Recurring Engineering and provide support to various Government applications. As it is proven out in government applications, we will pursue commercial opportunities, including licensing the software directly to commercial firms designing and developing CPSs. We have successfully done this in the past.

For example, Adventium developed the Framework for Analysis of Schedulability, Timing And Resources (FASTAR) tools for complex systems design over several NASA, Navy, and DARPA projects, and matured the technology under the Army Joint Multi-Role program. The JMR Technology Demonstrator is an Army Science and Technology program to get ready for Future Vertical Lift (FVL) acquisitions, expected to start in the early 2020s. Adventium Labs is supporting a series of Mission System Architecture Demonstrations (MSAD). The current one is the Architecture Implementation Process Demonstration (AIPD). Six companies have Technology Investment Agreements (TIAs) to explore a variety of standards, processes, methods and tools to improve development of mission systems for a family of FVL aircraft. In the near-term, we will present and demonstrate TEEE to a small DoD audience that is leading the JMR FVL avionics architecture development.

References

[1] Md Junaid Arafeen and Hyunsook Do. Test case prioritization using requirements-based clustering. In *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, pages 312–321. IEEE, 2013.

[2] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[3] Yanping Chen, Robert L Probert, and D Paul Sims. Specification-based regression test selection with risk analysis. In *Proceedings of the 2002 conference of the Centre for Advanced Studies on Collaborative research*, page 1. IBM Press, 2002.

[4] David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. The aetg system: An approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23(7):437–444, 1997.

[5] George Coker, Joshua Guttman, Peter Loscocco, Amy Herzog, Jonathan Millen, Brian O'Hanlon, John Ramsdell, Ariel Segall, Justin Sheehy, and Brian Sniffen. Principles of remote attestation. *International Journal of Information Security*, 10(2):63–81, June 2011.

[6] Tommaso Dreossi, Alexandre Donzé, and Sanjit A Seshia. Compositional falsification of cyber-physical systems with machine learning components. In *NASA Formal Methods Symposium*, pages 357–372. Springer, 2017.

[7] Peter Feiler, Bruce Lewis, and Steve Vestal. The SAE Avionics Architecture Description Language (AADL) Standard: A Basis for Model-Based Architecture-Driven Embedded Systems. In *Real-Time Applications Symposium Workshop on Model-Driven Embedded Systems*, 2003.

[8] Peter H Feiler, David P Gluch, and John J Hudak. The architecture analysis & design language (aadl): An introduction. Technical report, DTIC Document, 2006.

[9] Mats Grindal, Birgitta Lindström, Jeff Offutt, and Sten F Andler. An evaluation of combination strategies for test case selection. *Empirical Software Engineering*, 11(4):583–611, 2006.

[10] Jeffrey O Kephart and David M Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[11] Remo Lachmann, Michael Felderer, Manuel Nieke, Sandro Schulze, Christoph Seidl, and Ina Schaefer. Multi-objective black-box test case selection for system testing. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1311–1318. ACM, 2017.

[12] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, 2012.

[13] C Lott, Ashish Jain, and S Dalal. Modeling requirements for combinatorial software testing. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 1–7. ACM, 2005.

[14] Gary Mogyorodi. What is requirements-based testing? Technical report, Crosstalk, 2003.

[15] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[16] Robert Neches. Engineered Resilient Systems (ERS) S&T Priority Description And Roadmap, 2011.

[17] J. Penix and P. Alexander. Component reuse and adaptation at the specification level. In *8th Annual Workshop on Institutionalizing Software Reuse*, Ohio State University, Columbus, March 1997.
[18] John Penix and Perry Alexander. Toward automated component adaptation. In *Proceedings of the Ninth International Conference on Software Engineering and Knowledge Engineering*, pages 535–542. Knowledge Systems Institute, 1997.
[19] John Penix, Phillip Baraona, and Perry Alexander. Classification and retrieval of reusable components using semantic features. In *Proceedings of the 10th Knowledge-Based Software Engineering Conference*, pages 131–138, 1995.
[20] Jeff H Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, et al. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 87–102. ACM, 2009.
[21] Yuhua Qi, Xiaoguang Mao, and Yan Lei. Efficient automated program repair through fault-recorded testing prioritization. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 180–189. IEEE, 2013.
[22] Krishna Ranganathan, Murali Rangarajan, Perry Alexander, and Tom Regan. Automated test vector generation from rosetta requirements. In *VHDL International Users Forum Fall Workshop, 2000. Proceedings*, pages 51–58. IEEE, 2000.
[23] Ricardo J Rodríguez, José Merseguer, and Simona Bernardi. Modelling and analysing resilience as a security issue within uml. In *Proceedings of the 2nd international workshop on software engineering for resilient systems*, pages 42–51. ACM, 2010.
[24] Ana-Elena Rugina, Karama Kanoun, and Mohamed Kaâniche. A system dependability modeling framework using aadl and gspns. In *Architecting Dependable Systems IV*, pages 14–38. Springer, 2007.
[25] Society of Automotive Engineers. Architecture Analysis & Design Language (AADL). Aerospace Standard AS5506, 2004.
[26] Miruna Stoicescu, Jean-Charles Fabre, and Matthieu Roy. Architecting resilient computing systems: overall approach and open issues. In *International Workshop on Software Engineering for Resilient Systems*, pages 48–62. Springer, 2011.

List of Symbols, Abbreviations, and Acronyms

AADL	Architecture Analysis and Design Language
AIPD	Architecture Implementation Process Demonstration
API	Application Programming Interface
BRASS	Building Resource Adaptive Software Systems
CAMET	Curated Access to Model-based Engineering Tools
cP	Centipoise
CPS	Cyber Physical Systems
DSE	Design Space Explorer
DSEL	Domain Specific Embedded Language
FACE	Future Airborne Capability Environment
FASTAR	Framework for Analysis of Schedulability, Timing and Resources
FDN	Detected Failure
FRTP	Fault-Recorded Test Prioritization
FVL	Future Vertical Lift
GPIO	General Purpose Input/Output
GSPN	Generalized Stochastic Petri Nets
GUI	Graphical User Interface
IO	Input/Output
IP	Internet Protocol
JMR-TD	Joint Multi-Role Technology Demonstrator
LED	Light Emitting Diode
MBE	Model-based Engineering
MOEA	Multi-Objective Evolutionary Algorithm
MSAD	Mission System Architecture Demonstration
NAS	Network Attached Storage
OSATE	Open Source AADL Tool Environment
PCA	Patient Controlled Analgesia
PID	Proportional, Integral, Derivative
RBD	Reliability Block Diagram
RPM	Revolutions per Minute
SAVI	System Architecture Virtual Integration
SSA	Stimulus Synthesize Algorithm
SUT	System Under Test
TEEE	Toolkit for Evolving Ecosystem Envelopes
TIA	Technology Investment Agreements
TS	Test Suite
Ulps	microLiters per Second
UML	Universal Modeling Language